Introduction to Go

June 9, 2016

Rob Szewczyk
Developer, Next Century

What is Go?

"Go is an open source programming language that makes it easy to build simple, reliable, and efficient software."

golang.org (http://golang.org)

Go is:

- Statically typed
- Object oriented *
- Compiled
- Garbage collected
- Memory safe

History

Created at Google - design began in 2007.

Open source since 2009

Language stable since 2012 (Go 1)

Why?

- scalability
- simplicity

Language Tour

Example: Hello World Server

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprint(w, "Hello, World")
}

func main() {
    http.HandleFunc("/", handler)

    err := http.ListenAndServe(":7777", nil)
    if err != nil {
        log.Fatalln(err)
    }
}
```

Run

Packaging

Package Declaration

```
package main
```

Package dependencies

```
import (
    "fmt"
    "log"
    "net/http"
)
```

Package qualifier and exported package name

```
fmt.Fprint(...)
```

Declaration and Assignment

Variables

```
// short declaration and assignment (type is inferred)
err := http.ListenAndServe(":7777", nil)

// typed declaration
var err error
err = http.ListenAndServe(":7777", nil)
```

Functions

```
func handler(w http.ResponseWriter, r *http.Request) { ... }

// single and multiple return types
func ListenAndServe(string addr, handler http.Handler) error { ... }
func Marshal(i interface{}) (data []byte, err error) { ... }

// variadic
func Fprint(a ...interface{}) (int, error) { ... }
```

Example: xkcd server

```
package main

import (
    "log"
    "net/http"
    "strconv"

    "github.com/rszewczyk/gotalks/intro/xkcd-server/xkcd"
)

const doctypeTmpl = `<!doctype html>`

var xclient = xkcd.Client{
    URL: "http://xkcd.com",
}

func main() {
    // Fetch a single xkcd comic by id and display it
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        id, err := strconv.ParseInt(r.URL.Path[1:], 10, 0)
        if err != nil {
            http.Error(w, err.Error(), http.StatusBadRequest)
            return
        }

        comic, err := xclient.GetComic(int(id))
        if err != nil {
```

```
            http.Error(w, "", http.StatusNotFound)
            return
        }

        w.Write([]byte(doctypeTmpl))
        comic.WriteTo(w)
    })

    log.Fatalln(http.ListenAndServe(":7777", nil))
}
```

Run

More Packaging

"Go gettable" package names

```
"github.com/rszewczyk/gotalks/intro/xkcd-server/xkcd"
```

Exported identifier

```
xclient := xkcd.Client{
    URL: "http://xkcd.com",
}

// Client fetches comics from the configured xkcd URL
type Client struct {
    ...
}
```

Unexported identifier

```
// response represents the data from an xkcd API call
type response struct {
    ...
}
```

Structs

Type Declaration

```
type Client struct {
    // URL is an exported field (i.e. "public")
    URL string

    // these fields are unexported (i.e. "private")
    mu    sync.RWMutex //protects the cache
    cache map[int]response
    once  sync.Once
}
```

Initialization

```
client := &xkcd.Client{
    URL: "http://some.server.com,
}
```

Property Access

```
// from within the xkcd package
client.cache = make(map[int]response)
```

Declaring Types

User declared types have an underlying type (but are seperate distinct types). Examples:

```
type MyNumber int64
type HandlerFunc func(w http.ResponseWriter, r *http.Request)
type props map[string]string
type foo struct {}
```

Methods

Functions declared with a receiver

```
// receiver is a struct, but can be other user declared type
func (client *Client) GetComic(id int) (io.WriterTo, error) { ... }
```

Interfaces

Declared with a set of methods - Satisfied implicitly (but statically checked)

```
// from the io package
type WriterTo interface {
    WriterTo(w io.Writer) (int64, error)
}

// from the xkcd package
const comicTemplate = `<div>
<h3>%s</h3>
<img src="%s">
</div>
`

type apiResult struct {
    Title string `json:"safe_title"`
    Image string `json:"img"`
}

func (r *apiResult) WriteTo(w io.Writer) (int64, error) {
    n, err := fmt.Fprintf(w, comicTemplate, r.Title, r.Image)
    return int64(n), err
}
```

Concurrency

Communicating Sequential Processes (CSP)

Implemeted in Go with the following two concepts:
- go routines: very lightweight threads
- channels: typed pipes used to communicate and synchronize between goroutines

Go Routines

Create with the keyword go and a function invocation

```
go doSomeStuff()

go func() {
    doSomeStuff()
}()
```

go routines are maintained by the runtime which handles multiplexing them onto actual threads, switching them out for another when the running routine blocks

Channels

Declaring and initializing

```
var intChan chan int
intChan = make(chan int) // unbuffered

floatChan := make(chan float, 10) // buffered
```

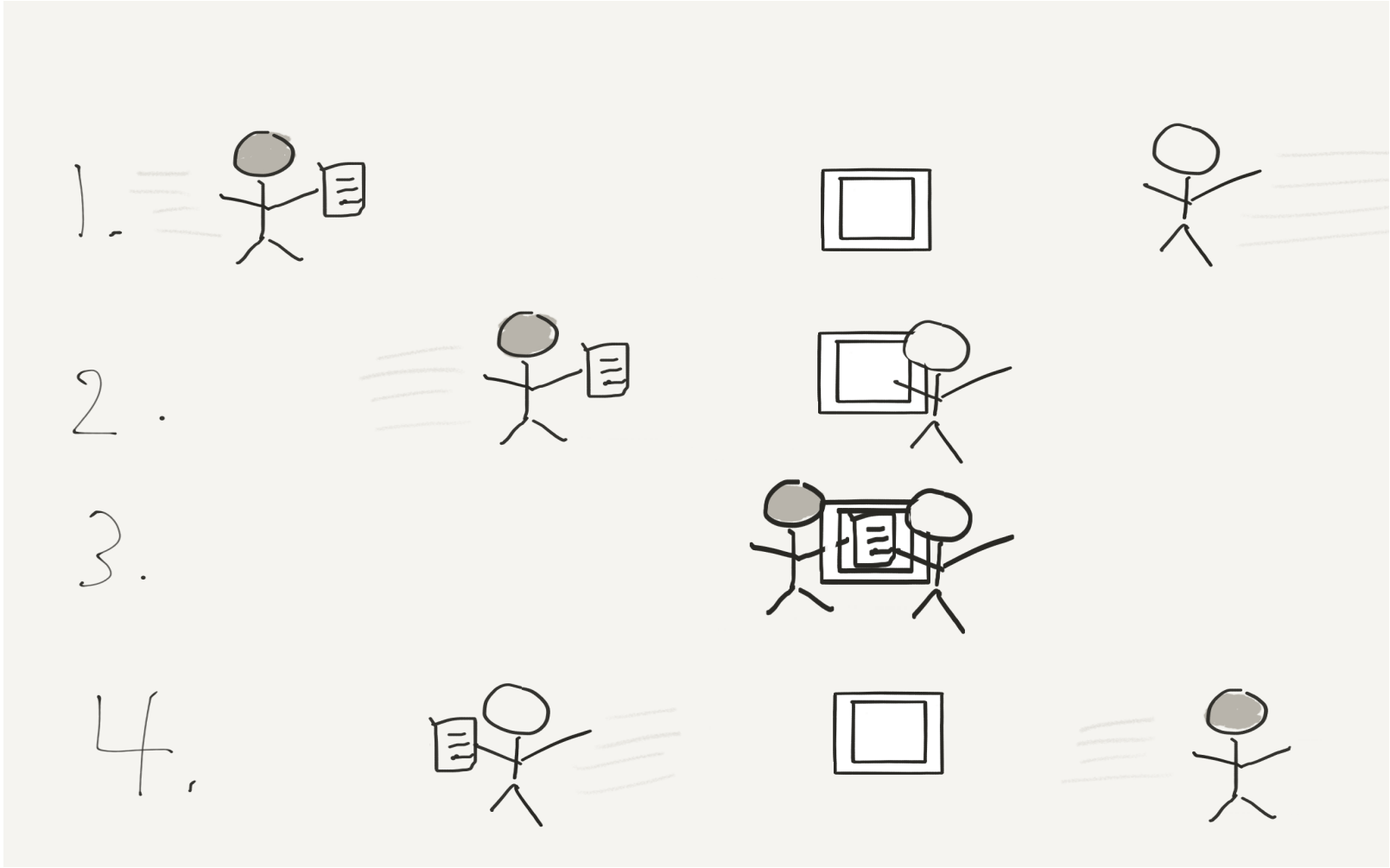
Send to a channel

```
intChan <- 42
```

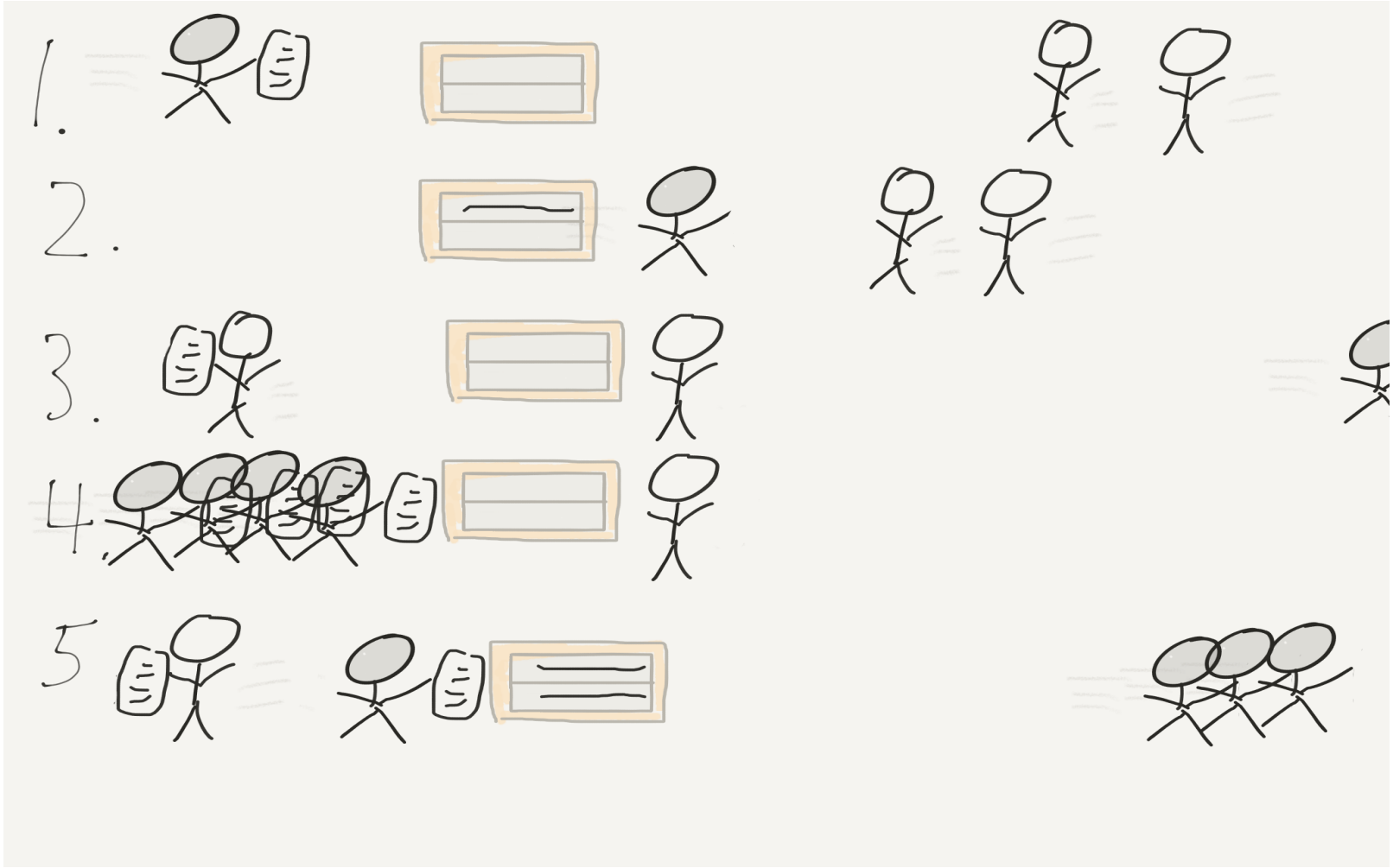
Receive from a channel

```
someFloat <-floatChan
```

Unbuffered Channels



Buffered Channels



Example: xkcd server

```
package main

import (
    "io"
    "log"
    "net/http"
    "strconv"
    "strings"

    "github.com/rszewczyk/gotalks/intro/xkcd-server/xkcd"
)

const doctypeTmpl = `<!doctype html>`

var xclient = xkcd.Client{
    URL: "http://xkcd.com",
}

func main() {
    // Fetch multiple xkcd comics by ids and display them (in correct order)
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        var (
            ids   []int
            chans []chan io.WriterTo
        )
        for _, s := range strings.Split(r.URL.Path[1:], ".") {
            id, err := strconv.ParseInt(s, 10, 0)
```

```
            if err != nil {
                http.Error(w, err.Error(), http.StatusBadRequest)
                return
            }
            ids = append(ids, int(id))
            chans = append(chans, make(chan io.WriterTo))
        }

        for i, id := range ids {
            go worker(id, chans[i])
        }

        w.Write([]byte(doctypeTmpl))

        for _, ch := range chans {
            comic := <-ch
            comic.WriteTo(w)
        }
    })

    log.Fatalln(http.ListenAndServe(":7776", nil))
}

func worker(id int, result chan io.WriterTo) {
    comic, _ := xclient.GetComic(id)
    result <- comic
}
```

Run

OOP in Go

Language support

Go lacks traditional OOP language features
- No inheritence
- No method overloading
- No classes or objects

However important OOP concepts are well supported
- encapsulation
- polymorphism

Encapsulation

Instead of using classes, types and behavior can be combined (using methods) and implementation can be hidden via packages and unexported identifiers.

```
type WriteCounter struct {
    ...
    count int
}

func (w *WriteCounter) Write(p []byte) (int, err) {
    ...
    count += numBytesWritten
    ...
}
```

Because there is no inheritence, encapsulation, in some ways, is actually stronger

Polymorphism

Iterfaces

```
type WriteCounter struct { ... }

func (w *WriteCounter) Write(p []byte) (int, err) { ...}

src, _ := os.Open("someFile")
dst := &WriteCounter{ ... }

io.Copy(dst, src)

// dst could also be
os.StdOut
http.ResponseWriter
bytes.Buffer
base64.Encoder
...
...
```

Polymorphism

Subtyping

Structs allow embedding via anonymous fields. The embedded type's methods are promoted to the embedding type. If the embedded type is an interface, then the embedding type will satify the is-a requirement for subtyping.

```
type WriteCounter struct {
    io.ReadWriter
    count int
}

func (w *WriteCounter) Write(p []byte) (int, err) {
    n, err := w.ReadWriter.Write(p)
    w.count += n
    return n, err
}
```

Note that even though this may look like inheritence and method overriding, it's not. It's still composition - the embedded ReadWriter is an exported field and is a value distinct from the containing value.

Tooling

Conclusion

Learning Go

tour.golang.org/welcome/1 (https://tour.golang.org/welcome/1)

play.golang.org/ (https://play.golang.org/)

golang.org/ref/spec (https://golang.org/ref/spec)

github.com/golang/go (https://github.com/golang/go)

Thank you

June 9, 2016

Rob Szewczyk
Developer, Next Century
rob.szew@gmail.com (mailto:rob.szew@gmail.com)